Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт Космических и Информационных Технологий
институт
Вычислительной Техники
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой
_____ А. И Легалов.
подпись инициалы, фамилия
« _____» _____ 20 ___ г.

БАКАЛАВРСКАЯ РАБОТА

09.03.01 Информатика и вычислительная техника
код и наименование направления

Библиотечная поддержка эволюционной разработки программных продуктов
Тема

Пояснительная записка

Руководитель	_____	зав. каф. ВТ, д.т.н.	А. И. Легалов
	подпись, дата	должность, ученая степень	инициалы, фамилия
Выпускник	_____		А. Е. Копцев
	подпись, дата		инициалы, фамилия
Нормоконтролер	_____	к.т.н., доцент	В.И. Иванов
	подпись, дата	должность, ученая степень	инициалы, фамилия

Красноярск 2016

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время существуют различные подходы к написанию компьютерных программ, определяемые как парадигмы программирования. Каждый из них определяет свой понятийный аппарат, инструменты, стиль формирования конструкций, написания кода, правила выполнения задач, которые программист ставит во время создания программы. Безусловно, можно подойти к определению парадигм, как философии мышления в программировании, но более оправданным следует считать их одним из инструментов, грамотное использование которых может повысить скорость и эффективность разработки.

Важным чертой современных языков программирования является то, что почти все из них являются мультипарадигменными. В связи с этим сделать однозначный вывод об эффективности той или иной парадигмы на основе высокой популярности языка не представляется возможным. Помимо этого, особенность любой парадигмы в том, что каждая из них позволяет наиболее эффективно решать лишь часть возникающих перед программистом задач. Поэтому изучение различных подходов к программированию достойно внимания.

Одной из достаточно молодых, но перспективных является процедурно-параметрическая (ПП) парадигма [1,2]. В ее основе лежит параметрический полиморфизм, позволяющий процедурам принимать и обрабатывать вариантные типы данных, без алгоритмического выбора альтернатив внутри этих процедур. Алгоритмическая обработка вариантов осуществляется с применением условных операторов или переключателей. Данный подход является развитием методов процедурного программирования и служит альтернативой объектно-ориентированному программированию.

Первым реальным воплощением ПП парадигмы стал язык программирования O2M, являющийся непосредственным расширением языка

Оберон-2 [3]. В нем сохраняется поддержка объектно-ориентированного стиля предшественника, но наряду с конструкциями данной парадигмы в нем реализованы дополнительные абстракции, поддерживающие процедурно-параметрическую парадигму.

Другой, более узкоспециализированной ветвью развития ПП парадигмы стала разработка экспериментального языка Alien [4] с синтаксисом, аналогичным синтаксису языка Оберон-2 и поддерживающим только процедурную парадигму. Поддержка ППП реализована путем добавления в процедурный язык конструкций, ориентированных на эволюционное [5] расширение типов данных и процедур.

В ходе работы с данными языками можно на практике ознакомиться с основными идеями ПП-парадигмы и развивать исследование параметрического подхода в различных направлениях. Помимо этого, предложены подходы написания ПП программ на традиционных языках программирования [1, 6]. Вместе с тем эффективность написания ПП программ с применением традиционных мультипарадигменных языков можно было бы повысить, если использовать дополнительную инструментальную поддержку с использованием метакомпиляторов или библиотек макроопределений и шаблонов.

Применение метакомпиляторов требует дополнительных усилий, сопоставимых с разработкой трансляторов средней сложности. Помимо этого, целесообразнее использовать их в связке с библиотеками макроопределений и шаблонов, ускоряющих генерацию нужных фрагментов кода (как в Qt [7]).

Наиболее быстрым для реализации и при этом достаточно эффективным подходом является использование библиотек макроопределений и шаблонов, формирующих для программистов дополнительную инструментальную прослойку. Ее применение позволяет повысить эффективность использования процедурно-параметрического стиля при написании кода на традиционных мультипарадигменных языках. Помимо этого, данную библиотеку можно использовать при построении генераторов кода как для метакомпиляторов, так и для экспериментальных компиляторов процедурно-параметрических языков

программирования. В связи с этим задача создания макробиблиотеки, обеспечивающей поддержку процедурно-параметрического стиля, является актуальной.

Целью данной работы является разработка библиотеки макросов, благодаря которой можно будет разрабатывать многомодульные, эволюционно-расширяемые программы на основе процедурно-параметрической парадигмы.

Для достижения поставленной цели в работе решались следующие задачи:

- провести изучение особенностей процедурно-параметрической парадигмы, рассмотрены особенности реализации ее основных конструкций на языке программирования C++;
- проанализировать подходы к построению процедурно-параметрических программ на языке программирования C++, рассмотрены возможности подхода при реализации различных элементарных вариантов расширения уже созданного кода;
- исследовать особенности использования процедурно-параметрического подхода при реализации традиционных паттернов ОО проектирования;
- на основе анализа кода, формируемого при реализации различных ситуаций, предложить макросы, включенные в состав библиотеки;
- провести перепроектирование примеров эволюционного расширения кода с применением разработанной библиотеки;
- сделать сравнительные оценки ранее написанного кода с кодом, использующим библиотеку макроопределений.

Результатом выполнения выпускной квалификационной работы является кроссплатформенная библиотека макроопределений, написанная на C++ и предназначенная для подключения к C++ проектам, проектирующимся в соответствии с процедурно-параметрической парадигмой.

В первом разделе рассматриваются основные особенности ППП. Анализируются способы реализации ПП конструкций на языке программирования C++.

Второй раздел посвящен описанию использования процедурно-

параметрических конструкций библиотеки на Примерах элементарного эволюционного расширения в различных направлениях. Приводится сравнение с возможностями расширения при процедурном и ОО программировании.

В третьем разделе рассматриваются особенности использования ПП стиля для реализации классических паттернов проектирования. Показано, как применение данного подхода позволяет избавиться от ряда недостатков, существующих при использовании паттернов.

В четвертом разделе предлагаются библиотека макроопределений, несколько комментариев к ней, а также примеры использования.

В пятом разделе приводится сравнительный анализ программ, написанных в процедурно-параметрическом стиле, как с библиотекой, так и без нее.

1 Особенности процедурно-параметрической парадигмы программирования

В работе используется следующая терминология, принятая при описании процедурно-параметрической парадигмы [1]:

1) Агрегаты данных – абстрактные типы данных, соответствующие структурам в C++;

2) Агрегаты процедур – программные конструкции, осуществляющие обработку агрегатов данных. Доступ к различным экземплярам данных осуществляется через один из элементов списка формальных параметров;

3) Параметрические обобщения (далее просто "обобщения") – программные объекты, предназначенные для группировки параметров, обеспечивающих однозначную связь с соответствующими специализациями. Необходимы для того, чтобы не смешивать логику специализации как сущности и специализации как объекта архитектуры. Могут быть многоуровневыми (специализации обобщения также являются обобщениями) и одноуровневыми (специализация состоит из основы обобщения и его специализаций);

4) Специализации обобщения (далее "специализации") – объединение стандартных типов данных, описывающих какую-либо сущность. Обработка экземпляров специализаций осуществляется только через обобщения, что обеспечивает корректный доступ и разделение логики. Основами специализаций называются агрегаты данных, обычно применяющиеся для описания программных объектов, входящих в параметрические обобщения в качестве специализаций;

5) Обобщающие параметрические процедуры (или просто параметрические процедуры) – программные объекты, поддерживающие механизм параметрического полиморфизма. Они не имеют аналогов в других парадигмах программирования. Их основная задача - описание параметрических аргументов и обработчика по умолчанию. Чаще всего пустые;

6) Обработчики параметрических специализаций – процедуры, осуществляющие непосредственную обработку параметрических специализаций. Используют обобщения, а не сами специализации в качестве параметров. Отличие обработчиков друг от друга заключается в наличии дополнительных признаков, идентифицирующих специализации, для обработки которых они предназначены.

Рассмотрим, как эти конструкции могут быть реализованы на языке C++.

В качестве примера основы специализации для фигуры можно рассмотреть прямоугольник. В коде ниже представлены абстракции данных, определяющие его, а также один из обработчиков данной специализации. Представленный обработчик выполняет создание и инициализацию прямоугольника по двум сторонам, переданным через входные параметры, а затем возвращает полученный экземпляр специализации.

```
// прямоугольник
struct Rectangle {
    int x, y; // ширина, высота
};

// Создание прямоугольника с инициализаций сторон
Rectangle* CreateRectangleAndInit(int x, int y) {
    Rectangle* pr = new Rectangle;
    Init(*pr, x, y);
    return pr;
}
```

Роль параметрического обобщения выполняет некоторая абстракция, которая объединяет все специализации. При рассмотрении геометрических фигур такой абстракцией служит обобщенная фигура, содержащая поле признака, связывающее ее с конкретной специализацией:

```
struct Figure {
    int mark;
};
```

Есть два пути реализации признака обобщения: статический и динамический. В первом случае признак регистрируется прямо в исходном коде

и этот признак у каждой фигуры постоянен при любом порядке работы с ними (пример признака прямоугольника):

```
const int rectangleMark = 1;
```

Во втором случае используется специальная функция, возвращающая автоматически зарегистрированный признак (пример ниже также для прямоугольника). Регистрация этого признака происходит при подключении файла специализации к проекту:

```
// Возвращает динамически сгенерированный признак
int GetRegMarkFigRectangle() {
    return regMark;
}
// Регистрирует фигуру-специализацию в обобщении
RegFigRectangle(const char* regInfo) {
// …
 regMark = GetSpecNumAndIncrement();
// …
}
```

Нетрудно понять, что случай с динамической реализацией признака предпочтительней – происходит избавление от магических чисел, представляющих значение признака, что облегчает восприятие существующего кода и его последующее расширение, особенно сторонними разработчиками. Дальнейшие примеры будут представлены, исходя из этих соображений.

Процедуры-обработчики обобщения представляют собой обобщенные функции, которые через механизм параметризации связываются с реализацией данной процедуры в специализации и вызывают ее.

```
// Функция-обработчик ввода для прямоугольника,
// подлежащая регистрации
void InFigRectangleValue(ifstream &ifst, Figure& f) {
 if(f.mark == GetRegMarkFigRectangle()) {
      In(ifst, static_cast<FigRectangle&>(f));
 }
 else throw;
 }
```

```
    // Регистратор функции ввода, представленной выше
    class RegInFigRectangleValue {
    public:
     RegInFigRectangleValue() {
          inFigureValue[GetRegMarkFigRectangle()] =
InFigRectangleValue;
     }
    };
    // При подключении файла с обработчиками создается объект,
    // в конструкторе которого происходит регистрация,
    // реализована через занесение указателя на функцию в массив
    RegInFigRectangleValue regInFigRectangleValue();
```

Выбор и вызов нужной функции (в нашем случае, ввода) осуществляется в обобщении следующим образом:

```
    void InFigureValue(ifstream &ifst, Figure& f) {
        //Выбор из массива функции ввода на основе признака фигуры
        InFigureValueFunc func = inFigureValue[f.mark];
        //Вызов функции ввода
        func(ifst, f);
    }
```

Полная версия представленных примеров находится в приложении А.

2 Моделирование процедурно-параметрического стиля для простых ситуаций расширения программ.

Всего было рассмотрено 7 случаев изменения функционала или структуры программ, которые служат для анализа возможностей различных стилей программирования. Эти случаи являются типичными и часто встречаются на практике[8]:

- расширение обобщений специализациями и, как следствие, расширение обрабатывающих их обобщающих процедур;
- добавление новых процедур, обеспечивающих дополнительную функциональность;
- добавление новых полей данных в существующие типы и изменение в соответствии с этим процедур, осуществляющих обработку измененных программных объектов;
- добавление новых процедур, предназначенных для обработки только одной из специализаций некоторого обобщения;
- создание нового обобщения на основе существующих специализаций;
- добавление в программу мультиметодов, осуществляющих обработку двух или более обобщенных параметров;
- изменение мультиметодов при добавлении новых специализаций в обобщения, используемые в качестве аргументов мультиметодов.

Сравнение процедурного, ОО и ПП подходов проведено в работе [9] и сделан вывод о преимуществах ПП-парадигмы в представленных примерах, поэтому далее будут рассмотрены предполагаемые варианты расширений в данной парадигме.

Модифицируется во всех случаях программа для работы с геометрическими фигурами [10], которые хранятся в едином контейнере.

Программа имеет следующие особенности:

1. Первоначальный комплект геометрических фигур состоит из треугольников и прямоугольников.

2. Заданные фигуры можно создавать, удалять, инициализировать, вводить и выводить из файлового потока.

В терминах процедурно-параметрической парадигмы обобщением в данной программе является класс фигуры, от которого наследуются фигуры-специализации (треугольник и прямоугольник). В файле обобщения определены также обработчики, осуществляющие параметризацию функций ввода, вывода, создания, удаления фигур. В файлах специализаций эти функции реализованы для конкретных фигур.

Полная версия данной программы расположена в приложении А.

2.1 Расширение обобщения добавлением новой специализации

Добавление специализации реализуется следующим образом: создаются новые файлы, описывающие структуру специализации и ее обработчики, среди которых есть автоматический регистратор специализации. При первом обращении к ней данный регистратор присваивает ей маркер, являющийся признаком данной специализации на протяжении работы программы.

Ниже представлен код заголовочного файла специализации:

```
// Структура Circle, определяющая содержимое круга
struct Circle {
    // Радиус круга
    int r;
};
// Инициализация существующего круга
void Init(Circle& c, int r);
// Создание круга с инициализаций сторон
Circle* CreateCircleAndInit(int r);
// Ввод данных в существующий круг из потока
```

```
void In(ifstream &ifst, Circle& c);
// Вывод данных о круге в поток
void Out(ofstream &ofst, Circle& c);
```

Сам регистратор специализации (некоторые незначительные детали кода опущены, полная версия в Приложении Б):

```
//Изначально маркер установлен в -1, это означает что
специализация не зарегистрирована и в программе не встречалась
int regMark = -1;
//   Класс,   обеспечивающий   регистрацию   данной   фигуры-
специализации
class RegFigCircle {
 public:
 RegFigCircle(const char* regInfo) {
      cout << regInfo << endl;
      regMark = GetSpecNumAndIncrement();
      cout << "     FigCircle was registered using number " <<
regMark << endl;
 }
};
RegFigCircle regFigCircle("Registration of: FigCircle");
```

Результатом данного расширения можно считать то, что экземпляры новой специализации были корректно зарегистрированы и добавлены в контейнер. Исходный код клиента не изменялся, т.к. не добавлялась новая функциональность.

2.2 Добавление новой процедуры, обеспечивающей дополнительную функциональность

Пусть для всех существующих фигур требуется добавить новую процедуру вычисления периметра.

Интересным оказалось то, что здесь расширение функционала потребовало редактирования сразу двух «уровней» - обобщения и специализации.

С последним уровнем все просто – для каждой фигуры был создан

отдельный файл содержащий единственную функцию Perimeter, которая в качестве аргумента принимает экземпляр специализации. Пример файла для треугольника:

```
// подключение специализации
#include "Triangle.h"
// вычисление периметра
double Perimeter(Triangle& t) {
    return t.a + t.b + t.c;
}
```

Что касается обобщения, то в его структуру был добавлен новый массив, содержащий указатели на функции вычисляющие периметр каждой фигуры, а также процедура, проверяющую тип фигуры, периметр которой запрашивается. Если тип фигуры и тип функции совпадают – вычисляется и возвращается периметр, иначе – бросается исключение.

На уровне контейнера создается еще один файл, в котором выполняется проверка добавленного функционала на примере вывода всех фигур с периметром больше 100:

```
#include "Container.h"
double Perimeter(Figure& f);

void FigurePerimeterTestOut(ofstream& ofst, Container& c) {
    for(int i = 0; i < c.size; i++) {
      double p = Perimeter(*(c.storage[i]));
      if(p > 100){
           ofst << i << ": ";
           ofst << p << endl;
      }
    }
}
```

В связи с добавлением в программу новой функциональности возникла необходимость изменить код клиента. Его специфика – использование всех передаваемых программных объектов (функций, классов) по своему

усмотрению. То есть, их вызов в любом порядке и в любом месте функции main.

Поэтому, если сервер, в ходе эволюционной разработки, обеспечивает клиента новыми функциями, то последний, для их использования в нужном контексте, должен измениться.

В данном примере был добавлен вызов функции, представленной выше, для одного из контейнеров:

```
ofst << "Figures Perimeters:" << endl;
FigurePerimeterTestOut(ofst, c);
```

Процедурно-параметрическая парадигма поддерживает эволюционный подход и для клиента, но в контексте простых задач это нецелесообразно – разработчику (возможно, стороннему) гораздо удобней варьировать функции произвольно, вставляя их в условные операторы, циклы, вызовы своих процедур.

Результатом данной работы можно считать то, что была добавлена несуществующая ранее возможность вычисления периметра для всех специализаций, протестирована на примере функции, листинг которой приведен выше. Эволюционный принцип разработки сервера соблюден. Клиент изменяется, так как изменяется выполняемый им набор функций. Файлы, расширяющие ранее написанный код, можно увидеть в приложении В.

2.3 Добавление новых полей данных в существующие типы и изменение в соответствии с этим процедур

Для существующих фигур нужно добавить новый параметр – цвет и редактирование в соответствии с ним процедур обработчиков (ввода-вывода, создания, инициализации)

В этом случае расширение заключается в добавлении декоратора как новой специализации. Доступ к нему поддерживается механизмом проверки признака специализации, по которому можно установить его тип [8].

Основа добавленной специализации выглядит следующим образом:

```
    // Структура FigColorDecorator, определяющая декоратор, задающий цвет фигуры и
    // указатель на следующую фигуру
    struct FigColorDecorator: Figure {
        // цвет задается строкой
        string color;
        // Указатель на фигуру
        Figure* pf;
    };
```

Процедуры-обработчики создают теперь дополнительный уровень, на котором выполняется новый функционал (например, ввод цвета). Затем уже происходит выбор нужной функции для конкретной специализации, обеспечивающей прежнее функционирование (выбор нужно сделать по той причине, что на вход процедурам декоратора подается экземпляр обобщения, а не специализации). Пример одной из процедур:

```
    // Ввод специализации фигуры-декоратора из потока  и следующей за ней фигуры
    void In(ifstream &ifst, FigColorDecorator& fcd) {
        ifst >> fcd.color;
        // После этого из файла вводится фигура, которая цепляется за декоратор
        fcd.pf = InFigure(ifst);
    }
```

Результатом данной работы можно считать то, что в программу было внесено структурное дополнение – цвет – и, в связи с этим, изменились обработчики новой специализации. Эволюционный принцип разработки соблюден. Файлы, расширяющие ранее написанный код, добавленный декоратором, расположены в приложении Г.

2.4 Добавление процедуры, предназначенной для обработки конкретных специализаций внутри обобщений

Добавим процедуру сортировки контейнера по периметру фигур.

Создадим новую единицу компиляции, которая будет содержать предварительное объявление обобщающей процедуры вычисления периметра, одна из конкретных реализаций которого выполняется в каждой итерации сортировки в зависимости от типа фигуры.

```
// Вычисление периметра обобщенной фигуры
double Perimeter(Figure& f);
// Сортировка вставками с перестановкой указателей на фигуры
void SortPerimeters(Container& c) {
    for(int i = 0; i < c.size-1; i++) {
        int iMin = i;
        for(int j = i + 1; j < c.size; j++) {
            if(Perimeter(*(c.storage[iMin])) >
                    Perimeter(*(c.storage[j]))) {
                iMin = j;
            }
        }
        Figure* pfTmp = c.storage[iMin];
        c.storage[iMin] = c.storage[i];
        c.storage[i] = pfTmp;
    }
}
```

Код представлен в приложении Д.

2.5 Добавление процедуры, осуществляющей выборочный вывод из контейнера

Добавим процедуру вывода только фигур-треугольников.

Для этого создадим новую функцию, которая выводит информацию о фигуре только в том случае, если та является треугольником:

```
void OnlyFigTriangleOut(ofstream& ofst, Figure& f) {
    if(GetFigMark(f) == GetRegMarkFigTriangle()) {
        Out(ofst, static_cast<FigTriangle&>(f));
    }
```

```
}
```

Для корректной работы написанного выше кода необходима функция, возвращающая признак фигуры. Невозможно использовать уже существующую реализацию, поскольку в целях предотвращения несанкционированного доступа она была обрамлена namespace'ом, а соответственно, может вызываться только в файле, где реализована. Поэтому необходимо создать такую функцию в отдельной единице компиляции и подключить ее к коду выше:

```
// Получение признака текущей фигуры
int GetFigMark(Figure& f) {
    // Проверка на всякий случай
    if(f.mark > -1) {
        return f.mark;
    }
    else {
        return -1;
    }
}
```

Результат – получена процедура, выполняющая вывод только фигур-треугольников. Также показан обход ситуации, когда область видимости одной из функций-обработчиков обобщения ограничена одним файлом, а ее функционал необходим в другом. Полный код представлен в Приложении Е.

2.6 Добавление мультиметодов

Регистрация и последующий выбор одной из комбинаций осуществляется через массив указателей на функции-обработчики конкретных отношений двух фигур.

```
MultimethodFunc multimethodFunc[10][10];
//  Функция, реализующая мультиметод для двух обобщенных фигур
void Multimethod(ofstream& ofst, Figure& f1, Figure& f2) {
    MultimethodFunc func = multimethodFunc[f1.mark][f2.mark];
```

```
    func(ofst, f1, f2);
}
//  Пример обработчик, вычисляющего отношения между
треугольником и прямоугольником
void TrianRectOut(ofstream &ofst, FigTriangle& ft1,
FigRectangle& fr2) {
    ofst << "We have Triangle and Rectangle";
    Out(ofst, ft1);
    Out(ofst, fr2);
}
```

Результат – добавленный мультиметод, задающий всевозможные отношения между двумя фигурами. Эволюционный принцип разработки соблюден. Полную версию кода, реализующего мультиметод, можно увидеть в приложении Ж.

2.7 Изменение мультиметодов при добавлении специализаций

Для выполнения задачи необходимо после добавления специализации реализовать код мультиметода для нее в отдельной единице компиляции и зарегистрировать каждый элемент мультиметода как отдельный обработчик специализации (код расширения представлен в приложении И)

Результат – гибкое расширение мультиметода после добавления специализации. Ранее написанный код мультиметода не затрагивается и без проблем взаимодействует с новым за счет заранее продуманного механизма регистрации – эволюционный метод разработки соблюден.

Таким образом, представленные на примере анализа простых ситуаций расширения функциональности и данных, при использовании ПП стиля, реализуются без изменения ранее написанного кода, что показывает гибкость рассматриваемого подхода.

3 Реализация паттернов с применением процедурно-параметрического стиля

Целью исследований является анализ возможностей процедурно-параметрического стиля, получаемых при реализации ситуаций, описываемых паттернами проектирования [11, 12].

3.1 Паттерн «Фасад»

В п.2.2 текущей работы было реализовано расширение функционала обобщений путем добавления нового обработчика, вычисляющего периметр. Если добавить еще одну подобную функцию, находящую значение, например, площади фигуры, а затем объединить их вызов в рамках новой функции вывода фигур, то полученный обработчик будет являться фасадом.

В итоге, вся реализация фасада сводится к добавлению и регистрации трех обработчиков обобщения: периметра, площади, расширенного вывода.

Вычисление площади делается по аналогии с вычислением периметра – добавление (в виде отдельных файлов) реализаций для каждой фигуры, а затем регистрация данных функций в отдельном параметрическом массиве Square, который будет осуществлять вызов нужной процедуры в зависимости от ее типа.

```
// В переменной регистрируются функции вычисления площади
SquareFunc squareFunc[10];

//  Функции вычисления площади обобщенной фигуры
double Square(Figure& f) {
    SquareFunc func = squareFunc[f.mark];
    return func(f);
}
```

Расширенный вывод добавляется точно так же:

```
// В переменной регистрируются функции расширенного вывода
ExtendedOutFigureFunc extendedOutFigure[10];

//  Функции расширенного вывода обобщенной фигуры
void ExtendedOutFigure(ofstream &ofst, Figure& f) {
    ExtendedOutFigureFunc func = extendedOutFigure[f.mark];
    func(ofst, f);
}
```

Ниже представлена непосредственная реализация расширенного вывода. Она и является фасадом.

```
void Out(ofstream &ofst, Figure& f) {
    ofst << "Extended Figure out:" << endl;
    OutFigure(ofst, *(fcd.pf));
    ofst << "    Perimeter is " << Perimeter(f) << endl;
    ofst << "    Square is " << Square(f) << endl;

}
```

Итог реализации данного паттерна – получение функции, выполняющей в результате одного вызова комплекс функций. Расширение данного паттерна не ограничено как в отношении самих фасадов, так и в отношении их составных частей (обработчиков обобщений), чего ОО парадигма без редактирования ранее написанного кода не позволяет.

3.2 Паттерн «Адаптер»

Возьмем в качестве примера постройку дома из треугольников, образующих крышу, и прямоугольников, образующих стены. Для корректной его постройки необходим адаптер, связывающий нахождение центра основания треугольника и центра верхней части стены.

Для обработки составных частей крыши – треугольников – напишем функцию вычисления медианы относительно каждой из сторон. Эта функция,

применимая только к специализации треугольник, делает интерфейсы существующих специализаций несовместимыми. Код функции ниже:

```
double GetMediana(Triangle& t){
 return sqrt((2*t.a*t.a+2*t.b*t.b-t.c*t.c)/4.0);
}
```

Часть крыши может быть описана следующим образом:

```
struct Roof { Triangle t; ... };
```

Для того, чтобы она правильно ложилась сверху на коробку дома, необходимо найти середину ее основания следующим образом

```
double GetCenter(Roof &r) { return GetMediana(r.t); }
```

Эта функция и будет являться реализацией адаптера. Ее добавление и регистрация делаются по аналогии с п.2.2 данной работы.

Гибкость процедурно-параметрического подхода здесь проявляется в том, что в любой момент можно добавлять для обработки адаптера Roof новые функции без его изменения.

3.3 Паттерн «Декоратор»

Реализован в п.2.3 данной работы.

При добавлении в фигуры новых функций обработки они без проблем добавляются в декоратор без изменения уже написанного кода. В отличие от ООП. То есть добавление сводится к п.2.2 данной работы.

3.4 Паттерн «Прокси»

Развивая тему фигур, предположим, что данные о специализациях хранятся во внешних файлах. Отдельно для каждой фигуры. В основном файле храним вместо них имена файлов.

Определим структуру прокси в отдельной единице компиляции, как в п.2.3 данной работы, где добавлялся декоратор:

```
struct FigProxy {
    // Имя файла, где содержится вся информация о фигуре
    string fileName;

    // Ссылка на оригинальную фигуру
    FigProxy *realFig;
    ...
};
```

По умолчанию, вместо экземпляров реальных фигур формируются прокси. Указатель на фигуру может быть пустой или реальный, если фигура в результате вызова какой-либо, связанной с ней, загрузилась в память.

Такой функцией, например, может стать вычисление периметра для. В п.2.2 было указано, как добавить такую функцию, отличие будет лишь в том, что ей передается в качестве аргумента прокси, а не экземпляр фигуры. Соответственно, перед вычислением периметра, функция должна проверить, инициализирован или нет указатель на реальную фигуру. Если нет, то нужно прочитать из файла данные, иначе – вычислить периметр путем переадресации на соответствующую функцию:

```
int Perimeter(FigProxy& fp) {
     if(fp.realFig==NULL)
          InFig(fp.fileName, fp.realFig);

     PerimeterFunc func = perimeterFunc[fp.realFig.mark];
     return func(fp.realFig);
}
```

Достоинства по сравнению с ООП те же, что и в предыдущем паттерне: расширение прокси на любой тип объектов и добавление новых функций обработки фигур без изменения уже написанного кода.

3.5 Паттерн «Компоновщик»

Продолжим развивать тему фигур.

Добавим дополнительную фигуру-специализацию Frame, которая включает в себя список фигур. Помимо этого добавляются дополнительные функции управления Add, Remove.

```
struct Frame{...};
struct FrameBuffer:Frame{
 vector<Frame*> c;
};
void Add(vector<Frame*> v, Frame f){
 v.push_back(f);
}
void Delete(vector<Frame*> v, int index){
 v.erase(v.begin() + index);
}
```

Далее определим фигуры-примитивы, а также функцию, собирающую из этих примитивов, например, дом:

```
struct FigTriangle:Frame {...}
struct FigCircle:Frame {...}

FrameBuffer* buildHouse()
{
 FrameBuffer * house = new FrameBuffer;
 for (int i=0; i<50; ++i)
      Add(house->c, new FigTriangle);
 for (int i=0; i<250; ++i)
      Add(house->c, new Rectangle);
 return house;
}
```

Эту функцию, например, можно использовать для построения 10 домов:

```
FrameBuffer* housePack = new FrameBuffer;
for (int i=0; i<10; i++)
 housePack->Add( buildHouse());
```

Для каждого примитива можно добавить функцию вычисления площади, а затем посчитать площадь всего FrameBuffer, который может состоять из любого количества FrameBuffer, FigTriangle, FigRectangle. Полученный функционал и

служит примером поведения компоновщика.

Плюсы процедурно-параметрического подхода в том, что за счет эволюционного добавления методов и полей можно расширять функционал и структуру компонозиции, опередялять новое поведение компоновщика без изменения ранее написанного кода.

3.6 Паттерн «Итератор»

Контекст задачи – постройка дома из фигур. Крышу и коробку делают разные строители из имеющегося у них набора фигур. У одного фигуры хранение фигур реализовано через массив, у другого – через список. Соответственно, у каждого из них собственные, отличающиеся методы для работы с этими структурами данных.

Руководителю стройки для запуска очередной части процесса создания дома необходимо задействовать все имеющиеся фигурки у строителей. Для этого ему понадобится итератор, чтобы, во-первых, скрыть реализацию структур данных, а во-вторых, чтобы упростить свою реализацию.

В процедурно-параметрическом контексте строителями будут специализации, а обобщением – начальник стройки.

```
// "строитель крыши" использует массив для хранения фигур
struct RoofBuilder {
 int size = 10;
 Figure setFigures[size];
}
// "строитель коробки" использует односвязный список
struct BoxBuilder {
    struct FigListNode{
        Figure fig;
        FigListNode *next;
    };
};
```

Основная функция начальника стройки – перебор всех имеющихся у

строителей фигур (и какие-либо действия с ними) после введения итератора будет выглядеть так:

```
void buildProcess(){
 // Создание итераторов для обоих строителей
 Iterator *rf, *bf;
 createIteratorRoofBuilder(rf);
 createIteratorBoxBuilder(bf);
 // Перебор разных реализаций при помощи одного интерфейса
 while(hasNextRoofBuilder(rf)) {
      Figure fig = (Figure)getNextRoofBuilder(rf);
      // используем в процессе постройки...
 }
 while(hasNextBoxBuilder(bf)) {
      Figure fig = (Figure)getNextBoxBuilder(bf);
      // используем в процессе постройки...
 }
}
```

Сам итератор представляет из себя структуру, содержащую текущую позицию в массиве данных, и функции для работы с каждым типом массива.

Плюс процедурно-параметрической реализации в том, что можно эволюционно расширять как сами итераторы, так и их обработчики.

3.7 Паттерн «Приспособленец»

В контексте фигур задачу можно трактовать следующим образом.

Для создания ограниченного алфавита предположим, что существует небольшой диапазон длин сторон для фигур (из этого следуют все возможные варианты типов фигур с различными сторонами). Сохраним все эти типы в ограниченном перечислении под соответствующими номерами.

```
enum FigType {Trian111, Trian112, Rect11, Rect12}
struct Fig {
 FigType type;
 int x_coord, y_coord;
```

```
}
```

Предположим, что хотим делать из фигур, например, стену дома, размещая на некоторой плоскости каждую из них (несколько сотен или более). Чтобы не хранить всю информацию о фигурах в загружаемом массиве, преобразуем параметры загружаемых из файлов фигур в индексы фигуры в перечислении.

В файле данные хранятся в виде:

```
координаты_x,y тип_фигуры стороны.
```

В массив фигур заносятся:

```
координаты_x,y номер_фигуры_с_данными_сторонами_в_enum.
```

Сильной стороной ПП реализации будет эволюционная расширяемость как самих типов фигур, так и их обработчиков.

3.8 Паттерн «Фабричный метод»

Программа о фигурах, описанная в п.2 данной работы выступает заготовкой для реализации данного паттерна. Ее нужно дополнить лишь функциями создания фигур-специализаций (добавление функций разобрано в п.2.2 текущей работы):

```
Figure* CreateTriangle() {
    Figure *f;
    f->mark = GetRegMarkFigTriangle();
    return f;
}

Figure* CreateRectangle() {
    Figure *f;
    f->mark = GetRegMarkFigRectangle();;
    return f;
}
```

Как видно из представленного фрагмента кода принцип сокрытия реализации конкретных фигур соблюден. Возможности эволюционного расширения, благодаря ПП реализации не ограничены как в плане создания

новых фабрик, так и расширения функционала текущих.

3.9 Паттерн «Прототип»

В качестве основы реализации использована программа, описанная в п.2. Добавляется и регистрируется параметрический обработчик, клонирующий фигуры:

```
// Клонирование фигуры
Figure* CloneFigure(Figure& f) {
    CloneFigureFunc func = cloneFigure[f.mark];
    Figure* pf = func(f);
    return pf;
}
```

Для каждой специализации определяется своя реализации процедуры клонирования, которая создает новый объект в соответствии с признаком обобщения и выставляет специализации корректный маркер. Пример для треугольника:

```
Figure *CloneFigRectangle(Figure& f) {
        FigRectangle *fr;
        fr->mark = GetRegMarkFigRectangle();
        return fr; }
```

Проведенный анализ показывает, что в ситуациях, описываемых паттернами ОО проектирования процедурно-параметрический подход позволяет решать задачи с использованием более простых типовых приемов, обеспечивающих помимо этого поддержку большей гибкости при создании эволюционно расширяемого ПО. Это подтверждает необходимость введения дополнительной инструментальной поддержки в традиционные мультипарадигменные языки с целью повысить эффективность использования в них ПП программирования.

4 Библиотека макроопределений

Предназначена для повышения отказоустойчивости ПП программ за счет вынесения ненадежных конструкций в отдельные единицы компиляции, а также для повышения удобства работы с процедурно-параметрическими конструкциями.

Мотивация создания конкретных макросов была простой – избавление от постоянного повторения наиболее часто встречающихся в предыдущих работах конструкций и упрощение написания кода для разработчика за счет инкапсулирования сложных конструкций.

Ниже представлены некоторые компоненты библиотеки:

1) Генератор основ специализаций. Создает при своем вызове минимальную заготовку для основы специализации с заданными именем, классом-родителем и типом поля.

```
#define CREATE_SPECIALIZATION(Name, BaseName, SpecName)
    struct Name : BaseName {
        SpecName _spec;
    };
```

2) Регистратор специализации. Выполняет регистрацию специализации (выдает ей маркер) с заданным именем. Обычно вызывается сразу же после генератора специализации. Каждая новая специализация создается с номером, большим на 1, чем последняя зарегистрированная. Безымянное пространство имен позволяет вызывать данный макрос только единожды в файле, где он подключается (файл специализации), тем самым обеспечивается примитивный контроль доступа.

```
#define REGISTER_SPECIALIZATION(SpecName, IncrFunc, DebugInfo)
    namespace
    {
        void InitRegMark##SpecName()
        {
```

```
            regMark##SpecName = IncrFunc();
        }
        ClassMarkRegistrar reg##SpecName(InitRegMark##SpecName,
DebugInfo);
    }
```

3) Регистратор обработчика обобщения. Выполняет регистрацию заданного метода в параметрическом массиве, который связывается через индекс с конкретной из специализаций. Шаблон используется для обеспечения универсальности типов параметрических контейнеров (например, функция вычисляющая периметр, должна возвращать целочисленное значение, а вывода на экран – не возвращает ничего)

```
class MethodRegistrar
{
public:
    template<typename TMethod>
    MethodRegistrar(
            TMethod container[],
            TMethod method,
            int index,
            const char* info = nullptr)
    {
        if (info != nullptr) std::cout << info << '\n';
        container[index] = method;
    }
};
#define REGISTER_METHOD(Container, Method, Mark, DebugInfo) \
    MethodRegistrar regMethod##Method(Container, Method, Mark,
DebugInfo);
```

Регистрация специализации после ее добавления выглядит следующим образом:

```
CREATE_REG_MARK_METHOD(FigRectangle);
REGISTER_SPECIALIZATION(FigRectangle,  GetSpecNumAndIncrement,
nullptr);
```

Макрос присваивает созданной FigRectangle порядковый номер, являющийся маркером специализации (затем увеличивает его для следующей специализации) и создает функцию-однострочник, возвращающую этот порядковый номер. Если макрос еще не зарегистрировал специализацию – вернется -1 вместо маркера (что нужно иметь ввиду при отладке ошибок, связанных с некорректной работой специализаций)

Регистрация параметрических обработчиков, обеспечивающих процедурно-параметрический полиморфизм, представлена ниже.

```
    REGISTER_METHOD(createFigureUseFileMark,
CreateFigRectangleUseFileMark, GetRegMarkFigRectangle(), nullptr);
    REGISTER_METHOD(inFigureValue,           InFigRectangleValue,
GetRegMarkFigRectangle(), nullptr);
    REGISTER_METHOD(outFigure,                  OutFigRectangle,
GetRegMarkFigRectangle(), nullptr);
    REGISTER_METHOD(deleteFigure,             DeleteFigRectangle,
GetRegMarkFigRectangle(), nullptr);
```

Макрос REGISTER_METHOD связывает обработчик (в данном случае их четыре – создание, ввод, вывод, удаление фигуры) с ее признаком, что при дальнейшем использовании позволит автоматически на основе признака специализации выбирать нужную функцию. Используя его, программист получает значительное сокращение исходного кода файлов специализаций по причине инкапсулирования технических сложных и объемных процедур регистрации специализаций.

Вызов небольшого, но удобного структурного макроса, связывающего интерфейс специализации с ней самой и с обобщением выглядит так:

```
CREATE_SPECIALIZATION(FigRectangle, Figure, Rectangle)
```

Его суть проста – создание структуры-наследника обобщения, который содержит экземпляр специализации. Использование макроса делает более четкой и понятной структуры зависимостей специализаций от обобщений и интерфейсов. Сокращение исходного кода в данном случае незначительное.

Библиотека позволила частично избавиться от длинного ручного

написания подобного кода и тем самым свести задачу добавления и регистрации ПП конструкций к набору простых действий. Еще одной отличительной чертой является то, что типовые задачи в ней выполняются единообразным способом. Также она повысила степень формализации процедурно-параметрических конструкций, что позволит в перспективе заменить ее метаобъектным компилятором.

Полный код библиотеки представлен в приложении Е.

5 Сравнение объема кода с применением макробиблиотеки и без нее

В таблице 5.1 был произведен подсчет строк, а в таблице 5.2 вычисление суммарного объема исходных файлов для программ, составленных во второй главе. Эти же программы были переписаны с использованием библиотеки макроопределений, описанной в предудыщей главе. Размер исходных файлов библиотеки при анализе не учитывался.

Таблица 5.1 – сравнение числа строк в исходном тексте программ до и после применения библиотеки макроопределений

Критерий сравнения	Число строк в коде без использования библиотеки	Число строк в коде с использованием библиотеки	Сокращение числа строк в исходных текстах, %
Добавление специализации	870	830	4.5
Добавление процедуры	850	805	5.3
Добавление полей	912	836	8.3
Добавление процедуры для специализаций	1023	954	6.7
Добавление процедуры для контейнера	741	725	2.2
Добавление мультиметодов	882	818	7.3
Изменение мультиметодов	1250	1154	7.7

Таблица 5.2 – сравнение объема файлов, содержащих исходный код, до и после применения библиотеки макроопределений

Критерий сравнения	Объем файлов без использования библиотеки, кб	Объем файлов с использованием библиотеки, кб	Сокращение числа строк в исходных текстах, %
Добавление специализации	37.3	34.9	6.43
Добавление процедуры	37.9	35.6	6,07

Окончание таблицы 5.2

Критерий сравнения	Объем файлов без	Объем файлов с	Сокращение

	использования библиотеки, кб	использованием библиотеки, кб	числа строк в исходных текстах, %
Добавление полей	36.7	33.8	8
Добавление процедуры для специализаций	40	36	10
Добавление процедуры для контейнера	31.3	30.7	2
Добавление мультиметодов	39.1	34.5	11.8
Изменение мультиметодов	54.1	47.4	12.4

Из представленных таблиц можно сделать вывод, что уменьшение исходного кода действительно имеет место. В случаях, где необходима регистрация в существующих параметрических массивах библиотека дает почти десятипроцентное сокращение.

Увеличить процент можно путем дальнейшего наращивания библиотеки. Например, создание функций-обработчиков специализаций, не регистрируемых в параметрических массивах и выполняющих роль интерфейса – создание, создание из файла, инициализация, создание и инициализация – можно вынести в отдельный макрос, включенный в библиотеку. Вызывать его нужно будет после регистрации специализации. Этот шаг позволит примерно на треть сократить файлы с реализацией обработчиков специализаций.

ЗАКЛЮЧЕНИЕ

В рамках выпускной квалификационной работы создана библиотека макросов, позволяющая повысить эффективно процедурно-параметрической парадигмы при разработке программ на C++.

В ходе решения поставленных задач были получены следующие результаты:

- Использование ПП стиля при программировании на мультипарадигменном языке показало возможность реализации ПП конструкций и повышение гибкости при эволюционном расширении программ по сравнению с процедурным и ОО подходами;
- При использовании ПП подхода для реализации ОО паттернов проектирования большинство ситуаций описывается добавлением или удалением процедурно-параметрических конструкций по достаточно простым типовым схемам, что позволяет больше внимания уделять технике программирования, а не изучению паттернов;
- Применение макроопределений позволяет повысить эффективность использования ПП стиля при программировании на современных мультипарадигменных языках;

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? [Электронный ресурс] : описание процедурно-параметрической парадигмы. — Режим доступа: http://www.softcraft.ru/ppp/pppfirst/pppfirst.pdf

2. Легалов, А. И. Процедурно-параметрическое программирование. / А. И. Легалов // Проблемы информатизации региона. ПИР-99: Сб. научных трудов пятой Всероссийской научно-практической конференции. Красноярск. КГТУ. - 1999. - С. 13-27.

3. Язык программирования O2M [Электронный ресурс] : документация языка O2M. - Режим доступа: http://www.softcraft.ru/ppp/o2m/o2mref.pdf/

4. Особенности языка процедурно-параметрического программирования [Электронный ресурс] : описание языка процедурно-параметрического программирования Alien. - Режим доступа: http://www.softcraft.ru/ppp/pppspecific/pppspecific.pdf

5. Легалов, А. И. Расширение модульной структуры программы за счет подключаемых модулей / А. И. Легалов, А. Я. Бовкун, И. А. Легалов // Доклады Академии наук высшей школы России, № 1 (14). – 2010. – С. 114-125.

6. Процедурно-параметрическое программирование [Электронный ресурс] : описание использования ПП-парадигмы на примере языка программирования C++. - Режим доступа: http://www.softcraft.ru/paradigm/ppp/ppp01.shtml

7. Использование мета-объектного компилятора (Meta-Object Compiler, moc) [Электронный ресурс] : обзор мета-объектного компилятора Qt. - Режим доступа: http://www.doc.crossplatform.ru/qt/4.6.x/moc.html

8. Легалов, А.И. Технология программирования. Процедурная и объектно-ориентированная парадигмы. Метод. указания по выполнению лабораторной работы № 1. / Красноярск: ИПЦ КГТУ, 2006. - 43 с.

9. Легалов, А.И. Эволюционное расширение программ при различных парадигмах программирования. / А. И. Легалов, И. А. Легалов, А. Ф. Солоха // Труды XVI Байкальской Всероссийской конференции «Информационные и математические технологии в науке и управлении». Часть III. - Иркутск: ИСЭМ СО РАН, 2011. ISBN 978-5-93908-094-1. - С. 42-49.

10. Легалов, А.И. Разнорукое программирование [Электронный ресурс] / А.И. Легалов // 2001. – Режим доступа: http://www.softcraft.ru/paradigm/dhp/index.shtml (дата обращения: 01.05.2016)

11. Гамма, Э. Приемы объектно-ориентированного пректирования. Паттерны проектирования: пер. с англ. / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес; пер. А. Слинкин. Санкт-Петербург: Питер, 2001. - 368 с.

12. Фримен, Э. Паттерны проектирования: пер. с англ. / Э. Фримен, Э. Фримен, К. Сьерра, Б. Бейтс. Санкт-Петербург.: Питер, 2012. - 656 с.

ПРИЛОЖЕНИЕ А

Заготовка приложения про фигуры, используемая для расширения

Main.cpp – Реализация клиента

```
#include <iostream>
using namespace std;
#include "Container.h"
int main(int argc, char* argv[])
{
    ifstream ifst(argv[1]);
    ofstream ofst(argv[2]);
    cout << "Start"<< endl;
    Container c;
    Init(c);
    In(ifst, c);
    ofst << "Filled container. " << endl;
    Out(ofst, c);
    ClearContainer(c);
    ofst << "Empty container. " << endl;
    Out(ofst, c);
    cout << "Stop"<< endl;
    return 0;
}
```

Container.h – Реализация функций работы с контейнером

```
#ifndef __Container__
#define __Container__
#include "Figure.h"
const int maxSize = 100;   // Размерность массива
struct Container {
    Figure* storage[maxSize];
    int size;
};
void Init(Container& c);
Container* CreateContainer(int x, int y);
void ClearContainer(Container& c);
void In(ifstream &ifst, Container& c);
void Out(ofstream &ofst, Container& c);
#endif
```

Container.cpp - Реализация функций, осуществляющих обработку контейнера

```
#include "Container.h"
void Init(Container& c) {c.size = 0;}
Container* CreateContainer(int x, int y) {
    Container* pc = new Container;
    Init(*pc);
    return pc;
}
void ClearContainer(Container& c) {
    for(int i = 0; i < c.size; i++) {
        delete c.storage[i];
    }
    Init(c);
}
void In(ifstream &ifst, Container& c) {
```

```
    while(!ifst.eof()) {
        if((c.storage[c.size] = InFigure(ifst)) != 0) {
            c.size++;
        }
    }
}
void Out(ofstream &ofst, Container& c) {
    ofst << "Container is containing " << c.size
           << " elements." << endl;
    for(int i = 0; i < c.size; i++) {
      ofst << i << ": ";
      OutFigure(ofst, *(c.storage[i]));
    }
}
```

Figure.h – Объявление функций-обработчиков обобщенной фигуры

```
#ifndef __Figure__
#define __Figure__
#include <fstream>
using namespace std;
struct Figure {
    // Признак
    int mark;
};
int GetSpecNumAndIncrement();
typedef Figure* (*CreateFigureFileMarkFunc)(int fileMark);
extern CreateFigureFileMarkFunc createFigureUseFileMark[];
typedef void (*InFigureValueFunc)(ifstream &ifst, Figure& f);
extern InFigureValueFunc inFigureValue[];
typedef void (*OutFigureFunc)(ofstream &ofst, Figure& f);
extern OutFigureFunc outFigure[];
typedef void (*DeleteFigureFunc)(Figure* f);
extern DeleteFigureFunc deleteFigure[];
Figure* CreateFigureUseFileMark(int fileMark);
void InFigureValue(ifstream &ifst, Figure& f);
Figure* InFigure(ifstream &ifst);
void OutFigure(ofstream &ofst, Figure& f);
void DeleteFigure(Figure* pf);
#endif
```

Figure.cpp – Реализация функций-обработчиков обобщенной фигуры

```
#include "Figure.h"
namespace {
    int specNumber = 0;
}
int GetSpecNumAndIncrement() {
    return specNumber++;
}
CreateFigureFileMarkFunc createFigureUseFileMark[10];
InFigureValueFunc inFigureValue[10];
OutFigureFunc outFigure[10];
DeleteFigureFunc deleteFigure[10];
Figure* CreateFigureUseFileMark(int fileMark) {
    Figure* pf;
    for(int i = 0; i < specNumber; i++) {
        pf = createFigureUseFileMark[i](fileMark);
        if(pf != 0) return pf;
    }
    return 0;
}
void InFigureValue(ifstream &ifst, Figure& f) {
    InFigureValueFunc func = inFigureValue[f.mark];
```

```
    func(ifst, f);
}
Figure* InFigure(ifstream &ifst) {
    int fileMark;
    ifst >> fileMark;
    Figure* pf = CreateFigureUseFileMark(fileMark);
    if(pf == 0) return 0;
    InFigureValue(ifst, *pf);
    return pf;
}
void OutFigure(ofstream &ofst, Figure& f) {
    OutFigureFunc func = outFigure[f.mark];
    func(ofst, f);
}
void DeleteFigure(Figure* pf) {
    DeleteFigureFunc func = deleteFigure[pf->mark];
    func(pf);
}
```

FigRectangle.h – Объявление функций, осуществляющих обработку прямоугольника

```
#ifndef __FigRectangle__
#define __FigRectangle__
#include "Figure.h"
#include "Rectangle.h"
struct FigRectangle: Figure {
    Rectangle r;
};
int GetRegMarkFigRectangle();
void Init(FigRectangle& fr, int x, int y);
FigRectangle* CreateFigRectangle();
FigRectangle* CreateFigRectangleAndInit(int x, int y);
Figure* CreateFigRectangleUseFileMark(int fileMark);
void In(ifstream &ifst, FigRectangle& fr);
void Out(ofstream &ofst, FigRectangle& fr);
void DeleteFigRectangle(FigRectangle* pft);
#endif
```

FigRectangle.cpp – Реализация функций, осуществляющих обработку прямоугольника

```
#include <iostream>
#include "FigRectangle.h"
namespace {
    int regMark = -1;
}
int GetRegMarkFigRectangle() {
    return regMark;
}
namespace {
    class RegFigRectangle {
    public:
        RegFigRectangle(const char* regInfo) {
            cout << regInfo << endl;
            regMark = GetSpecNumAndIncrement();
            cout << "FigRectangle was registered using number" << regMark;
        }
    };
    RegFigRectangle regFigRectangle("Registration of: FigRectangle");
}
void Init(FigRectangle& fr, int x, int y) {
    fr.mark = GetRegMarkFigRectangle();
    Rectangle& r = fr.r;
    Init(r, x, y);
}
```

```
FigRectangle* CreateFigRectangle() {
    FigRectangle* pfr = new FigRectangle;
    Init(*pfr, 0, 0);
    return pfr;
}
FigRectangle* CreateFigRectangleAndInit(int x, int y) {
    FigRectangle* pfr = CreateFigRectangle();
    Init(*pfr, x, y);
    return pfr;
}
void In(ifstream &ifst, FigRectangle& fr) {
    Rectangle& r = fr.r;
    In(ifst, r);
}
void Out(ofstream &ofst, FigRectangle& fr) {
    ofst << "Rectangle is as Specialization of Figure: x = " << fr.r.x << ", y =
" << fr.r.y << endl;
}
void DeleteFigRectangle(FigRectangle* pfr) {
    delete pfr;
}
namespace {
    Figure* CreateFigRectangleUseFileMark(int fileMark) {
        if(fileMark == 1) {
            FigRectangle* pfr = CreateFigRectangle();
            return  pfr;
        }
        return 0;
    }
    class RegCreateFigRectangleUseFileMark {
    public:
        RegCreateFigRectangleUseFileMark(const char* regInfo);
    };
 RegCreateFigRectangleUseFileMark::RegCreateFigRectangleUseFileMark(const char*
regInfo) {
        cout << regInfo << endl;
        createFigureUseFileMark[GetRegMarkFigRectangle()] =
CreateFigRectangleUseFileMark;
        cout << "    createFigureUseFileMark[" << GetRegMarkFigRectangle() << "]
= CreateFigRectangleUseFileMark" << endl;
    }
    RegCreateFigRectangleUseFileMark
regCreateFigRectangleUseFileMark("Registration of
CreateFigRectangleUseFileMark");
}
namespace {
    void InFigRectangleValue(ifstream &ifst, Figure& f) {
        if(f.mark == GetRegMarkFigRectangle()) {
            In(ifst, static_cast<FigRectangle&>(f));
        }
    }
    class RegInFigRectangleValue {
    public:
        RegInFigRectangleValue(const char* regInfo) {
            cout << regInfo << endl;
            inFigureValue[GetRegMarkFigRectangle()] = InFigRectangleValue;
            cout << "    inFigureValue[" << GetRegMarkFigRectangle() << "] =
InFigRectangleValue" << endl;
        }
    };
    RegInFigRectangleValue regInFigRectangleValue("Registration of
InFigRectangleValue");
}
namespace {
```

```
    void OutFigRectangle(ofstream &ofst, Figure& f) {
        // Проверка на всякий случай
        if(f.mark == GetRegMarkFigRectangle()) {
            Out(ofst, static_cast<FigRectangle&>(f));
        }

    }
    class RegOutFigRectangle {
    public:
        RegOutFigRectangle(const char* regInfo) {
            cout << regInfo << endl;
            outFigure[GetRegMarkFigRectangle()] = OutFigRectangle;
            cout << "    outFigure[" << GetRegMarkFigRectangle() << "] =
OutFigRectangle" << endl;
        }
    };
    RegOutFigRectangle regOutFigRectangle("Registration of OutFigRectangle");
}
namespace {
    void DeleteFigRectangleSkin(Figure* pf) {
        if(pf->mark == GetRegMarkFigRectangle()) {
            DeleteFigRectangle(static_cast<FigRectangle*>(pf));
        }
    }
    class RegDeleteFigRectangleSkin {
    public:
        RegDeleteFigRectangleSkin(const char* regInfo) {
            cout << regInfo << endl;
            deleteFigure[GetRegMarkFigRectangle()] = DeleteFigRectangleSkin;
            cout << "    outFigure[" << GetRegMarkFigRectangle() << "] =
DeleteFigRectangleSkin" << endl;
        }
    };
    RegDeleteFigRectangleSkin regDeleteFigRectangleSkin("Registration of
DeleteFigRectangleSkin");
}
```

FigTriangle.h – Объявление интерфейсных функций, осуществляющих обработку треугольника

```
#ifndef __FigTriangle__
#define __FigTriangle__
#include "Figure.h"
#include "Triangle.h"
struct FigTriangle: Figure {
    Triangle t;
};
int GetRegMarkFigTriangle();
void Init(FigTriangle& ft, int a, int b, int c);
FigTriangle* CreateFigTriangle();
FigTriangle* CreateFigTriangleAndInit(int a, int b, int c);
Figure* CreateFigTriangleUseFileMark(int fileMark);
void In(ifstream &ifst, FigTriangle& ft);
void Out(ofstream &ofst, FigTriangle& ft);
void DeleteFigTriangle(FigTriangle* pft);
#endif
```

FigTriangle.cpp - Реализация интерфейсных функций, осуществляющих обработку треугольника

```
#include <iostream>
#include "FigTriangle.h"
namespace {
    int regMark = -1;
```

```
}
int GetRegMarkFigTriangle() {
    return regMark;
}
namespace {
    class RegFigTriangle {
    public:
        RegFigTriangle(const char* regInfo) {
            cout << regInfo << endl;
            regMark = GetSpecNumAndIncrement();
            cout << "FigTriangle was registered using number " << regMark;
        }
    };
    RegFigTriangle regFigTriangle("Registration of: FigTriangle");
}
void Init(FigTriangle& ft, int a, int b, int c) {
    ft.mark = GetRegMarkFigTriangle();
    Triangle& t = ft.t;
    Init(t, a, b, c);
}
FigTriangle* CreateFigTriangle() {
    FigTriangle* pft = new FigTriangle;
    Init(*pft, 0, 0, 0);
    return pft;
}
FigTriangle* CreateFigTriangleAndInit(int a, int b, int c) {
    FigTriangle* pft = CreateFigTriangle();
    Init(*pft, a, b, c);
    return pft;
}
void In(ifstream &ifst, FigTriangle& ft) {
    Triangle& t = ft.t;
    In(ifst, t);
}
void Out(ofstream &ofst, FigTriangle& ft) {
    ofst << "Triangle is as Specialization of Figure: a = " << ft.t.a
         << ", b = " << ft.t.b << ", c = " << ft.t.c << endl;
}
void DeleteFigTriangle(FigTriangle* pft) {
    delete pft;
}
namespace {
    Figure* CreateFigTriangleUseFileMark(int fileMark) {
        if(fileMark == 2) {
            FigTriangle* pft = CreateFigTriangle();
            return  pft;
        }
        return 0;
    }
    class RegCreateFigTriangleUseFileMark {
    public:
        RegCreateFigTriangleUseFileMark(const char* regInfo);
    };
    RegCreateFigTriangleUseFileMark::RegCreateFigTriangleUseFileMark(const char*
regInfo) {
        cout << regInfo << endl;
        createFigureUseFileMark[GetRegMarkFigTriangle()] =
CreateFigTriangleUseFileMark;
        cout << "    createFigureUseFileMark[" << GetRegMarkFigTriangle() << "]
= CreateFigTriangleUseFileMark" << endl;
    }
regCreateFigTriangleUseFileMark("Registration of CreateFigTriangleUseFileMark");
}
namespace {
```

```
    void InFigTriangleValue(ifstream &ifst, Figure& f) {
        if(f.mark == GetRegMarkFigTriangle()) {
            In(ifst, static_cast<FigTriangle&>(f));
        }
    }
    class RegInFigTriangleValue {
    public:
        RegInFigTriangleValue(const char* regInfo) {
            cout << regInfo << endl;
            inFigureValue[GetRegMarkFigTriangle()] = InFigTriangleValue;
            cout << "    inFigureValue[" << GetRegMarkFigTriangle() << "] =
InFigTriangleValue" << endl;
        }
    };
    RegInFigTriangleValue regInFigTriangleValue("Registration of
InFigTriangleValue");
}
namespace {
    void OutFigTriangle(ofstream &ofst, Figure& f) {
        if(f.mark == GetRegMarkFigTriangle()) {
            Out(ofst, static_cast<FigTriangle&>(f));
        }
    }
    class RegOutFigTriangle {
    public:
        RegOutFigTriangle(const char* regInfo) {
            cout << regInfo << endl;
            outFigure[GetRegMarkFigTriangle()] = OutFigTriangle;
            cout << "    outFigure[" << GetRegMarkFigTriangle() << "] =
OutFigTriangle" << endl;
        }
    };
    RegOutFigTriangle regOutFigTriangle("Registration of OutFigTriangle");
}
namespace {
    void DeleteFigTriangleSkin(Figure* pf) {
        if(pf->mark == GetRegMarkFigTriangle()) {
            DeleteFigTriangle(static_cast<FigTriangle*>(pf));
        }
    }
    class RegDeleteFigTriangleSkin {
    public:
        RegDeleteFigTriangleSkin(const char* regInfo) {
            cout << regInfo << endl;
            deleteFigure[GetRegMarkFigTriangle()] = DeleteFigTriangleSkin;
            cout << "    outFigure[" << GetRegMarkFigTriangle() << "] =
DeleteFigTriangleSkin" << endl;
        }
    };
    RegDeleteFigTriangleSkin regDeleteFigTriangleSkin("Registration of
DeleteFigTriangleSkin");
}
```

Rectangle.h – Объявление структуры прямоугольника и функций, осуществляющих его обработку

```
#ifndef __Rectangle__
#define __Rectangle__
struct Rectangle {
    int x;
    int y;
};
void Init(Rectangle& r, int x, int y);
Rectangle* CreateRectangleAndInit(int x, int y);
void In(ifstream &ifst, Rectangle& r);
```

```
void Out(ofstream &ofst, Rectangle& r);
#endif
```

Rectangle.cpp - Реализация функций, осуществляющих обработку прямоугольника

```
#include "Rectangle.h"
void Init(Rectangle& r, int x, int y) {
    r.x = x;
    r.y = y;
}
Rectangle* CreateRectangleAndInit(int x, int y) {
    Rectangle* pr = new Rectangle;
    Init(*pr, x, y);
    return pr;
}
void In(ifstream &ifst, Rectangle& r) {
    ifst >> r.x >> r.y;
}
void Out(ofstream &ofst, Rectangle& r) {
    ofst << "Rectangle: x = " << r.x << ", y = " << r.y << endl;
}
```

Triangle.h – Объявление структуры треугольника и функций, осуществляющих его обработку

```
#ifndef __Triangle__
#define __Triangle__
struct Triangle {
    int a;
    int b;
    int c;
};
void Init(Triangle& t, int a, int b, int c);
Triangle* CreateTriangleAndInit(int a, int b, int c);
void In(ifstream &ifst, Triangle& t);
void Out(ofstream &ofst, Triangle& t);
#endif
```

Triangle.cpp – Реализация функций, осуществляющих обработку треугольника

```
#include "Triangle.h"
void Init(Triangle& t, int a, int b, int c) {
    t.a = a;
    t.b = b;
    t.c = c;
}
Triangle* CreateTriangleAndInit(int a, int b, int c) {
    Triangle* pt = new Triangle;
    Init(*pt, a, b, c);
    return pt;
}
void In(ifstream &ifst, Triangle& t) {
    ifst >> t.a >> t.b >> t.c;
}
void Out(ofstream &ofst, Triangle& t) {
    ofst << "Triangle: a = " << t.a << ", b = " << t.b << ", c = " << t.c <<
endl;
}
```

ПРИЛОЖЕНИЕ Б

Расширение обобщения добавлением новой специализации

Добавляется файл Circle.h – Объявление структуры круга и функций, осуществляющих его обработку

```
#ifndef __Circle__
#define __Circle__
struct Circle {
    int r;
};
void Init(Circle& c, int r);
Circle* CreateCircleAndInit(int r);
void In(ifstream &ifst, Circle& c);
void Out(ofstream &ofst, Circle& c);
#endif
```

Добавляется файл Circle.cpp - Реализация функций, осуществляющих обработку круга

```
#include "Circle.h"
void Init(Circle& c, int r) {
    c.r = r;
}
Circle* CreateCircleAndInit(int r) {
    Circle* pc = new Circle;
    Init(*pc, r);
    return pc;
}
void In(ifstream &ifst, Circle& c) {
    ifst >> c.r;
}
void Out(ofstream &ofst, Circle& c) {
    ofst << "Circle: r = " << c.r << endl;
}
```

Добавляется файл FigCircle.h - Объявление интерфейсных функций, осуществляющих обработку круга

```
#ifndef __FigCircle__
#define __FigCircle__
#include "Figure.h"
#include "Circle.h"
struct FigCircle: Figure {
    Circle c;
};
int GetRegMarkFigCircle();
void Init(FigCircle& fc, int r);
FigCircle* CreateFigCircle();
FigCircle* CreateFigCircleAndInit(int r);
Figure* CreateFigCircleUseFileMark(int fileMark);
void In(ifstream &ifst, FigCircle& fc);
void Out(ofstream &ofst, FigCircle& fc);
void DeleteFigCircle(FigCircle* pfc);
#endif
```

Добавляется файл FigCircle.cpp – Реализация интерфейсных функций, осуществляющих обработку круга

```
#include <iostream>
```

```
#include "FigCircle.h"
namespace {
    int regMark = -1;
}
int GetRegMarkFigCircle() {
    return regMark;
}
namespace {
    class RegFigCircle {
    public:
        RegFigCircle(const char* regInfo) {
            cout << regInfo << endl;
            regMark = GetSpecNumAndIncrement();
            cout << "FigCircle was registered using number " << regMark;
        }
    };
    RegFigCircle regFigCircle("Registration of: FigCircle");
}
void Init(FigCircle& fc, int r) {
    fc.mark = GetRegMarkFigCircle();
    Circle& c = fc.c;
    Init(c, r);
}
FigCircle* CreateFigCircle() {
    FigCircle* pfc = new FigCircle;
    Init(*pfc, 0);
    return pfc;
}
FigCircle* CreateFigCircleAndInit(int r) {
    FigCircle* pfc = CreateFigCircle();
    Init(*pfc, r);
    return pfc;
}
void In(ifstream &ifst, FigCircle& fc) {
    Circle& c = fc.c;
    In(ifst, c);
}
void Out(ofstream &ofst, FigCircle& fc) {
    ofst << "Circle is as Specialization of Figure: r = " << fc.c.r << endl;
}
void DeleteFigCircle(FigCircle* pfc) {
    delete pfc;
}
namespace {
    Figure* CreateFigCircleUseFileMark(int fileMark) {
        if(fileMark == 3) {
            FigCircle* pfc = CreateFigCircle();
            return  pfc;
        }
        return 0;
    }
    class RegCreateFigCircleUseFileMark {
    public:
        RegCreateFigCircleUseFileMark(const char* regInfo);
    };
RegCreateFigCircleUseFileMark::RegCreateFigCircleUseFileMark(const char*
regInfo) {
        cout << regInfo << endl;
        createFigureUseFileMark[GetRegMarkFigCircle()] =
CreateFigCircleUseFileMark;
        cout << "    createFigureUseFileMark[" << GetRegMarkFigCircle() << "] =
CreateFigCircleUseFileMark" << endl;
    }
    RegCreateFigCircleUseFileMark regCreateFigCircleUseFileMark("Registration of
```

```
CreateFigCircleUseFileMark");
}
namespace {
    void InFigCircleValue(ifstream &ifst, Figure& f) {
        if(f.mark == GetRegMarkFigCircle()) {
            In(ifst, static_cast<FigCircle&>(f));
        }
    }
    class RegInFigCircleValue {
    public:
        RegInFigCircleValue(const char* regInfo) {
            cout << regInfo << endl;
            inFigureValue[GetRegMarkFigCircle()] = InFigCircleValue;
            cout << "    inFigureValue[" << GetRegMarkFigCircle() << "] =
InFigCircleValue" << endl;
        }
    };
    RegInFigCircleValue regInFigCircleValue("Registration of InFigCircleValue");
}
namespace {
    void OutFigCircle(ofstream &ofst, Figure& f) {
        if(f.mark == GetRegMarkFigCircle()) {
            Out(ofst, static_cast<FigCircle&>(f));
        }
    }
    class RegOutFigCircle {
    public:
        RegOutFigCircle(const char* regInfo) {
            cout << regInfo << endl;
            outFigure[GetRegMarkFigCircle()] = OutFigCircle;
            cout << "    outFigure[" << GetRegMarkFigCircle() << "] =
OutFigCircle" << endl;
        }
    };
    RegOutFigCircle regOutFigCircle("Registration of OutFigCircle");
}
namespace {
    void DeleteFigCircleSkin(Figure* pf) {
        if(pf->mark == GetRegMarkFigCircle()) {
            DeleteFigCircle(static_cast<FigCircle*>(pf));
        }
    }
    class RegDeleteFigCircleSkin {
    public:
        RegDeleteFigCircleSkin(const char* regInfo) {
            cout << regInfo << endl;
            deleteFigure[GetRegMarkFigCircle()] = DeleteFigCircleSkin;
            cout << "    outFigure[" << GetRegMarkFigCircle() << "] =
DeleteFigCircleSkin" << endl;
        }
    };
    RegDeleteFigCircleSkin regDeleteFigCircleSkin("Registration of
DeleteFigCircleSkin");
}
```

ПРИЛОЖЕНИЕ В

Добавление новой процедуры, обеспечивающей дополнительную функциональность

Добавляется файл RectPerimeter.cpp – Реализация функции вычисления периметра прямоугольника

```
#include "Rectangle.h"
double Perimeter(Rectangle& r) {
    return 2.0 * (r.x + r.y);
}
```

Добавляется файл TrianPerimeter.cpp – Реализация функции вычисления периметра прямоугольника

```
#include "Triangle.h"
double Perimeter(Triangle& t) {
    return t.a + t.b + t.c;
}
```

Добавляется файл FigRectPerimeter.cpp – Реализация интерфейса функции, вычисляющей периметр прямоугольника

```
#include <iostream>
#include "FigRectangle.h"
#include "FigPerimeter.h"
double Perimeter(Rectangle& r);
double Perimeter(FigRectangle& fr) {
    Rectangle& r = fr.r;
    return Perimeter(r);
}
namespace {
    double PerimeterFigRectangle(Figure& f) {
        if(f.mark == GetRegMarkFigRectangle()) {
            return Perimeter(static_cast<FigRectangle&>(f));
        }
    }
    class RegPerimeterFigRectangle {
    public:
        RegPerimeterFigRectangle(const char* regInfo) {
            cout << regInfo << endl;
            perimeterFunc[GetRegMarkFigRectangle()] = PerimeterFigRectangle;
            cout << "    perimeterFunc[" << GetRegMarkFigRectangle() << "] =
PerimeterFigRectangle" << endl;
        }
    };
    RegPerimeterFigRectangle regPerimeterFigRectangle("Registration of
PerimeterFigRectangle");
}
```

Добавляется файл FigTrianPerimeter.cpp – Реализация интерфейса функции, вычисляющей периметр треугольника

```
#include <iostream>
#include "FigTriangle.h"
#include "FigPerimeter.h"

double Perimeter(Triangle& t);
```

```
double Perimeter(FigTriangle& ft) {
    Triangle& t = ft.t;
    return Perimeter(t);
}
namespace {
    double PerimeterFigTriangle(Figure& f) {
        if(f.mark == GetRegMarkFigTriangle()) {
            return Perimeter(static_cast<FigTriangle&>(f));
        }
    }
    class RegPerimeterFigTriangle {
    public:
        RegPerimeterFigTriangle(const char* regInfo) {
            cout << regInfo << endl;
            perimeterFunc[GetRegMarkFigTriangle()] = PerimeterFigTriangle;
            cout << "    perimeterFunc[" << GetRegMarkFigTriangle() << "] =
PerimeterFigTriangle" << endl;
        }
    };
    RegPerimeterFigTriangle regPerimeterFigTriangle("Registration of
PerimeterFigTriangle");
}
```

Добавляется файл FigPerimeter.h – Объявление функции, вычисляющей периметр обобщенной фигуры

```
#ifndef __FigPerimeter__
#define __FigPerimeter__
#include "Figure.h"
typedef double (*PerimeterFunc)(Figure& f);
extern PerimeterFunc perimeterFunc[];
double Perimeter(Figure& f);
#endif
```

Добавляется файл FigPerimeter.cpp – Реализация функции, вычисляющей периметр обобщенной фигуры

```
#include "FigPerimeter.h"
PerimeterFunc perimeterFunc[10];
double Perimeter(Figure& f) {
    PerimeterFunc func = perimeterFunc[f.mark];
    return func(f);
}
```

Добавляется файл ContainerTestPerimeter.cpp – Реализация функции обхода фигур с выводом их периметров

```
#include "Container.h"
double Perimeter(Figure& f);
void FigurePerimeterTestOut(ofstream& ofst, Container& c) {
    for(int i = 0; i < c.size; i++) {
            double p = Perimeter(*(c.storage[i]));
            if(p > 100){
                  ofst << i << ": ";
                  ofst << p << endl;
            }
    }
}
```

ПРИЛОЖЕНИЕ Г

Добавление новых полей данных в существующие типы и изменение в соответствии с этим процедур

Добавляется файл FigColorDecorator.h – Объявление функций, осуществляющих обработку декоратора

```
#ifndef __FigColorDecorator__
#define FigColorDecorator__
#include "Figure.h"
#include <string>
struct FigColorDecorator: Figure {
    string color;
    Figure* pf;
};
int GetRegMarkFigColorDecorator();
void Init(FigColorDecorator& fcd, string color, Figure* pf);
FigColorDecorator* CreateFigColorDecorator();
FigColorDecorator* CreateFigColorDecoratorAndInit(string color, Figure* pf);
Figure* CreateFigColorDecoratorUseFileMark(int fileMark);
void In(ifstream &ifst, FigColorDecorator& fcd);
void Out(ofstream &ofst, FigColorDecorator& fcd);
void DeleteFigColorDecorator(FigColorDecorator* pfcd);
#endif
```

Добавляется файл FigColorDecorator.cpp – Реализация функций, осуществляющих обработку декоратора

```
#include <iostream>
#include "FigColorDecorator.h"
namespace {
    int regMark = -1;
}
int GetRegMarkFigColorDecorator() {
    return regMark;
}
namespace {
    class RegFigColorDecorator {
    public:
        RegFigColorDecorator(const char* regInfo) {
            cout << regInfo << endl;
            regMark = GetSpecNumAndIncrement();
            cout << "    FigColorDecorator was registered using number " <<
regMark << endl;
        }
    };

    RegFigColorDecorator regFigColorDecorator("Registration of:
FigColorDecorator");
}
void Init(FigColorDecorator& fcd, string color, Figure* pf) {
    fcd.mark = GetRegMarkFigColorDecorator();
    fcd.color = color;
    fcd.pf = pf;
}
FigColorDecorator* CreateFigColorDecorator() {
    FigColorDecorator* pfcd = new FigColorDecorator;
    Init(*pfcd, "", 0);
```

```
    return pfcd;
}
FigColorDecorator* CreateFigColorDecoratorAndInit(string color, Figure* pf) {
    FigColorDecorator* pfcd = CreateFigColorDecorator();
    Init(*pfcd, color, pf);
    return pfcd;
}
void In(ifstream &ifst, FigColorDecorator& fcd) {
    ifst >> fcd.color;
    fcd.pf = InFigure(ifst);
}
void Out(ofstream &ofst, FigColorDecorator& fcd) {
    ofst << "Colored Figure:" << endl;
    ofst << "    ";
    OutFigure(ofst, *(fcd.pf));
    ofst << "    Color is " << fcd.color << endl;
}
void DeleteFigColorDecorator(FigColorDecorator* pfcd) {
    delete pfcd;
}
namespace {
    Figure* CreateFigColorDecoratorUseFileMark(int fileMark) {
        if(fileMark == 13) {
            FigColorDecorator* pfcd = CreateFigColorDecorator();
            return  pfcd;
        }
        return 0;
    }
    class RegCreateFigColorDecoratorUseFileMark {
    public:
        RegCreateFigColorDecoratorUseFileMark(const char* regInfo);
    };
RegCreateFigColorDecoratorUseFileMark::RegCreateFigColorDecoratorUseFileMark(con
st char* regInfo) {
        cout << regInfo << endl;
        createFigureUseFileMark[GetRegMarkFigColorDecorator()] =
CreateFigColorDecoratorUseFileMark;
        cout << "    createFigureUseFileMark[" << GetRegMarkFigColorDecorator()
<< "] = CreateFigColorDecoratorUseFileMark" << endl;
    }
    RegCreateFigColorDecoratorUseFileMark
regCreateFigColorDecoratorUseFileMark("Registration of
CreateFigColorDecoratorUseFileMark");
}
namespace {
    void InFigColorDecoratorValue(ifstream &ifst, Figure& f) {
        if(f.mark == GetRegMarkFigColorDecorator()) {
            In(ifst, static_cast<FigColorDecorator&>(f));
        }
    }
    class RegInFigColorDecoratorValue {
    public:
        RegInFigColorDecoratorValue(const char* regInfo) {
            cout << regInfo << endl;
            inFigureValue[GetRegMarkFigColorDecorator()] =
InFigColorDecoratorValue;
            cout << "    inFigureValue[" << GetRegMarkFigColorDecorator() << "]
= InFigColorDecoratorValue" << endl;
        }
    };
    RegInFigColorDecoratorValue regInFigColorDecoratorValue("Registration of
InFigColorDecoratorValue");
}
namespace {
```

```
    void OutFigColorDecorator(ofstream &ofst, Figure& f) {
        if(f.mark == GetRegMarkFigColorDecorator()) {
            Out(ofst, static_cast<FigColorDecorator&>(f));
        }
    }
    class RegOutFigColorDecorator {
    public:
        RegOutFigColorDecorator(const char* regInfo) {
            cout << regInfo << endl;
            outFigure[GetRegMarkFigColorDecorator()] = OutFigColorDecorator;
            cout << "    outFigure[" << GetRegMarkFigColorDecorator() << "] =
OutFigColorDecorator" << endl;
        }
    };
    RegOutFigColorDecorator regOutFigColorDecorator("Registration of
OutFigColorDecorator");
}
namespace {
    void DeleteFigColorDecoratorSkin(Figure* pf) {
        if(pf->mark == GetRegMarkFigColorDecorator()) {
DeleteFigColorDecorator(static_cast<FigColorDecorator*>(pf));
        }
    }
    class RegDeleteFigColorDecoratorSkin {
    public:
        RegDeleteFigColorDecoratorSkin(const char* regInfo) {
            cout << regInfo << endl;
            deleteFigure[GetRegMarkFigColorDecorator()] =
DeleteFigColorDecoratorSkin;
            cout << "    outFigure[" << GetRegMarkFigColorDecorator() << "] =
DeleteFigColorDecoratorSkin" << endl;
        }
    };
    RegDeleteFigColorDecoratorSkin regDeleteFigColorDecoratorSkin("Registration
of DeleteFigColorDecoratorSkin");
}
```

ПРИЛОЖЕНИЕ Д

Добавление процедуры, предназначенной для обработки конкретных специализаций внутри обобщений

Добавляется файл ContainerSorting.cpp – Реализация функцию сортировки фигур по возрастанию их периметров

```
#include "Container.h"
double Perimeter(Figure& f);
void SortPerimeters(Container& c) {
    for(int i = 0; i < c.size-1; i++) {
        int iMin = i;
        for(int j = i + 1; j < c.size; j++) {
            if(Perimeter(*(c.storage[iMin])) > Perimeter(*(c.storage[j]))) {
                iMin = j;
            }
        }
        Figure* pfTmp = c.storage[iMin];
        c.storage[iMin] = c.storage[i];
        c.storage[i] = pfTmp;
    }
}
```

Изменяется файл Main.cpp – Добавление вызова новых функций ввиду расширения функционала клиента

```
#include <iostream>
using namespace std;
#include "Container.h"
void FigurePerimeterTestOut(ofstream& ofst, Container& c);
void SortPerimeters(Container& c);
int main(int argc, char* argv[])
{
    ifstream ifst(argv[1]);
    ofstream ofst(argv[2]);

    cout << "Start"<< endl;
    Container c;
    Init(c);
    In(ifst, c);
    ofst << "Filled container. " << endl;
    Out(ofst, c);
    ofst << "Figures Perimeters:" << endl;
    FigurePerimeterTestOut(ofst, c);
    ofst << "Figures sorting with using perimeter values" << endl;
    SortPerimeters(c);
    ofst << "Sorted container. " << endl;
    Out(ofst, c);
    ofst << "Figures Perimeters after sorting:" << endl;
    FigurePerimeterTestOut(ofst, c);
    ClearContainer(c);
    ofst << "Empty container. " << endl;
    Out(ofst, c);
    cout << "Stop"<< endl;
    return 0;
}
```

ПРИЛОЖЕНИЕ Е

Добавление процедуры, осуществляющей выборочный вывод из контейнера

Добавляется файл GetFigMark.cpp – Реализация функции, возвращающей значение признака текущей фигуры

```
#include "Figure.h"
int GetFigMark(Figure& f) {
    // Проверка на всякий случай
    if(f.mark > -1) {
        return f.mark;
    }
    else {
        return -1;
    }
}
```

Добавляется файл FigTriangleOutOnly.cpp – Реализация функции, вычисляющей периметр треугольника

```
#include "FigTriangle.h"
int GetFigMark(Figure& f);
void OnlyFigTriangleOut(ofstream& ofst, Figure& f) {
    if(GetFigMark(f) == GetRegMarkFigTriangle()) {
        Out(ofst, static_cast<FigTriangle&>(f));
    }
}
```

Добавляется файл ContainerTestOutOnlyTriangles.cpp – Реализация функции обхода фигур с выводом только треугольников-специализаций

```
#include "Container.h"
void OnlyFigTriangleOut(ofstream& ofst, Figure& f);
void OnlyFigTriangleTestOut(ofstream& ofst, Container& c) {
    for(int i = 0; i < c.size; i++) {
      Figure* pf = c.storage[i];
      OnlyFigTriangleOut(ofst, *pf);
    }
}
```

ПРИЛОЖЕНИЕ Ж

Добавление мультиметода

Добавляется файл FigMm.h – Объявление функций, реализующих мультиметод для двух обобщенных фигур

```
#ifndef __FigMm__
#define __FigMm__
#include "Figure.h"
typedef void (*MultimethodFunc)(ofstream& ofst, Figure& f1, Figure& f2);
extern MultimethodFunc multimethodFunc[][10];
void Multimethod(ofstream& ofst, Figure& f1, Figure& f2);
#endif
```

Добавляется файл FigMm.cpp – Реализация функции, реализующей мультиметод для двух обобщенных фигур

```
#include "FigMm.h"
MultimethodFunc multimethodFunc[10][10];
void Multimethod(ofstream& ofst, Figure& f1, Figure& f2) {
    MultimethodFunc func = multimethodFunc[f1.mark][f2.mark];
    func(ofst, f1, f2);
}
```

Добавляется файл FigTrianRectMm.cpp - Реализация обработчиков специализаций мультиметода для всех комбинаций треугольника и прямоугольника

```
#include "FigTriangle.h"
#include "FigRectangle.h"
#include "FigMm.h"
#include <iostream>
void TrianTrianOut(ofstream &ofst, FigTriangle& ft1, FigTriangle& ft2) {
    ofst << "This is two Triangles" << endl;
    Out(ofst, ft1);
    Out(ofst, ft2);
}
void TrianRectOut(ofstream &ofst, FigTriangle& ft1, FigRectangle& fr2) {
    ofst << "We have Triangle and Rectangle" << endl;
    Out(ofst, ft1);
    Out(ofst, fr2);
}
void RectTrianOut(ofstream &ofst, FigRectangle& fr1, FigTriangle& ft2) {
    ofst << "The first figure is Rectangle and second is Triangle" << endl;
    Out(ofst, fr1);
    Out(ofst, ft2);
}
void RectRectOut(ofstream &ofst, FigRectangle& fr1, FigRectangle& fr2) {
    ofst << "Rectangle + Rectangle = Two Rectangles" << endl;
    Out(ofst, fr1);
    Out(ofst, fr2);
}
void MmTrianTrianSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
    if(f1.mark == GetRegMarkFigTriangle() && f2.mark == GetRegMarkFigTriangle())
return TrianTrianOut(ofst, static_cast<FigTriangle&>(f1),
static_cast<FigTriangle&>(f2));
}
void MmTrianRectSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
```

```
    if(f1.mark == GetRegMarkFigTriangle() && f2.mark ==
GetRegMarkFigRectangle()) {
        return TrianRectOut(ofst, static_cast<FigTriangle&>(f1),
static_cast<FigRectangle&>(f2));
    }
}
void MmRectTrianSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
    if(f1.mark == GetRegMarkFigRectangle() && f2.mark ==
GetRegMarkFigTriangle()) {
        return RectTrianOut(ofst, static_cast<FigRectangle&>(f1),
static_cast<FigTriangle&>(f2));
    }
}
void MmRectRectSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
    if(f1.mark == GetRegMarkFigRectangle() && f2.mark ==
GetRegMarkFigRectangle()) {
        return RectRectOut(ofst, static_cast<FigRectangle&>(f1),
static_cast<FigRectangle&>(f2));
    }
}
namespace {
    class Register {
    public:
        Register(const char* regInfo);
    };
    Register::Register(const char* regInfo) {
        cout << regInfo << endl;
multimethodFunc[GetRegMarkFigTriangle()][GetRegMarkFigTriangle()] =
MmTrianTrianSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigTriangle() << "][" <<
GetRegMarkFigTriangle() << "] = MmTrianTrianSpecOut" << endl;
multimethodFunc[GetRegMarkFigTriangle()][GetRegMarkFigRectangle()] =
MmTrianRectSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigTriangle() << "][" <<
GetRegMarkFigRectangle() << "] = MmTrianRectSpecOut" << endl;
multimethodFunc[GetRegMarkFigRectangle()][GetRegMarkFigTriangle()] =
MmRectTrianSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigRectangle() << "][" <<
GetRegMarkFigTriangle() << "] = MmRectTrianSpecOut" << endl;
multimethodFunc[GetRegMarkFigRectangle()][GetRegMarkFigRectangle()] =
MmRectRectSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigRectangle() << "][" <<
GetRegMarkFigRectangle() << "] = MmRectRectSpecOut" << endl;
    }
    Register trianRegisterPerimeter("Registration of: Different combination of
Triangle and Rectangle in  Multimethod");
}
```

Добавляется файл ContainerMmTestOut.cpp – Реализация, функции осуществляющей перебор парных комбинаций для всех элементов контейнера

```
#include "Container.h"
void Multimethod(ofstream& ofst, Figure& f1, Figure& f2);
void MultimethodTestOut(ofstream& ofst, Container& c) {
    for(int i = 0; i < c.size; i++) {
        for(int j = 0; j < c.size; j++) {
            Figure* pf1 = c.storage[i];
            Figure* pf2 = c.storage[j];
            ofst << "[" << i << "][" << j << "]: ";
            Multimethod(ofst, *pf1, *pf2);
        }
    }}
```

ПРИЛОЖЕНИЕ И

Изменение мультиметода при добавлении специализаций

Добавляется файл FigTrianRectMm.cpp – Реализация обработчиков специализаций мультиметода для всех комбинаций треугольника и прямоугольника

```
#include "FigTriangle.h"
#include "FigRectangle.h"
#include "FigMm.h"
#include <iostream>
void TrianTrianOut(ofstream &ofst, FigTriangle& ft1, FigTriangle& ft2) {
    ofst << "This is two Triangles" << endl;
    Out(ofst, ft1);
    Out(ofst, ft2);
}
void TrianRectOut(ofstream &ofst, FigTriangle& ft1, FigRectangle& fr2) {
    ofst << "We have Triangle and Rectangle" << endl;
    Out(ofst, ft1);
    Out(ofst, fr2);
}
void RectTrianOut(ofstream &ofst, FigRectangle& fr1, FigTriangle& ft2) {
    ofst << "The first figure is Rectangle and second is Triangle" << endl;
    Out(ofst, fr1);
    Out(ofst, ft2);
}
void RectRectOut(ofstream &ofst, FigRectangle& fr1, FigRectangle& fr2) {
    ofst << "Rectangle + Rectangle = Two Rectangles" << endl;
    Out(ofst, fr1);
    Out(ofst, fr2);
}
void MmTrianTrianSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
    if(f1.mark == GetRegMarkFigTriangle() && f2.mark == GetRegMarkFigTriangle())
{
        return TrianTrianOut(ofst, static_cast<FigTriangle&>(f1),
static_cast<FigTriangle&>(f2));
    }
}
void MmTrianRectSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
    if(f1.mark == GetRegMarkFigTriangle() && f2.mark ==
GetRegMarkFigRectangle()) {
        return TrianRectOut(ofst, static_cast<FigTriangle&>(f1),
static_cast<FigRectangle&>(f2));
    }
}
void MmRectTrianSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
    // Проверка на всякий случай
    if(f1.mark == GetRegMarkFigRectangle() && f2.mark ==
GetRegMarkFigTriangle()) {
        return RectTrianOut(ofst, static_cast<FigRectangle&>(f1),
static_cast<FigTriangle&>(f2));
    }
}
void MmRectRectSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
    if(f1.mark == GetRegMarkFigRectangle() && f2.mark ==
GetRegMarkFigRectangle()) {
        return RectRectOut(ofst, static_cast<FigRectangle&>(f1),
static_cast<FigRectangle&>(f2));
    }
```

```
}
namespace {
    class Register {
    public:
        Register(const char* regInfo);
    };
    Register::Register(const char* regInfo) {
        cout << regInfo << endl;
multimethodFunc[GetRegMarkFigTriangle()][GetRegMarkFigTriangle()] =
MmTrianTrianSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigTriangle() << "][" <<
GetRegMarkFigTriangle() << "] = MmTrianTrianSpecOut" << endl;
multimethodFunc[GetRegMarkFigTriangle()][GetRegMarkFigRectangle()] =
MmTrianRectSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigTriangle() << "][" <<
GetRegMarkFigRectangle() << "] = MmTrianRectSpecOut" << endl;
multimethodFunc[GetRegMarkFigRectangle()][GetRegMarkFigTriangle()] =
MmRectTrianSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigRectangle() << "][" <<
GetRegMarkFigTriangle() << "] = MmRectTrianSpecOut" << endl;
multimethodFunc[GetRegMarkFigRectangle()][GetRegMarkFigRectangle()] =
MmRectRectSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigRectangle() << "][" <<
GetRegMarkFigRectangle() << "] = MmRectRectSpecOut" << endl;
    }
    Register trianRegisterPerimeter("Registration of: Different combination of
Triangle and Rectangle in  Multimethod");
}
```

Добавляется файл FigTrianRectMmAdd.cpp – Реализация обработчиков специализаций мультиметода для всех комбинаций треугольника, прямоугольника и круга

```
#include "FigTriangle.h"
#include "FigRectangle.h"
#include "FigCircle.h"
#include "FigMm.h"
#include <iostream>
void TrianCircOut(ofstream &ofst, FigTriangle& ft1, FigCircle& fc2) {
    ofst << "Triangle & Circle Company" << endl;
    Out(ofst, ft1);
    Out(ofst, fc2);
}
void RectCircOut(ofstream &ofst, FigRectangle& fr1, FigCircle& fc2) {
    ofst << "We have Rectangle & Circle after its" << endl;
    Out(ofst, fr1);
    Out(ofst, fc2);
}
void CircTrianOut(ofstream &ofst, FigCircle& fc1, FigTriangle& ft2) {
    ofst << "This Circle is before Triangle" << endl;
    Out(ofst, fc1);
    Out(ofst, ft2);
}
void CircRectOut(ofstream &ofst, FigCircle& fc1, FigRectangle& fr2) {
    ofst << "Circle + Rectanle = 4 Angles" << endl;
    Out(ofst, fc1);
    Out(ofst, fr2);
}
void CircCircOut(ofstream &ofst, FigCircle& fc1, FigCircle& fc2) {
    ofst << "2 * Circle = 8" << endl;
    Out(ofst, fc1);
    Out(ofst, fc2);
}
void MmTrianCircSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
```

```
    if(f1.mark == GetRegMarkFigTriangle() && f2.mark == GetRegMarkFigCircle()) {
        return TrianCircOut(ofst, static_cast<FigTriangle&>(f1),
static_cast<FigCircle&>(f2));
    }
}
void MmRectCircSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
    if(f1.mark == GetRegMarkFigRectangle() && f2.mark == GetRegMarkFigCircle())
{
        return RectCircOut(ofst, static_cast<FigRectangle&>(f1),
static_cast<FigCircle&>(f2));
    }
}
void MmCircTrianSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
    if(f1.mark == GetRegMarkFigCircle() && f2.mark == GetRegMarkFigTriangle()) {
        return CircTrianOut(ofst, static_cast<FigCircle&>(f1),
static_cast<FigTriangle&>(f2));
    }
}
void MmCircRectSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
    if(f1.mark == GetRegMarkFigCircle() && f2.mark == GetRegMarkFigRectangle())
{
        return CircRectOut(ofst, static_cast<FigCircle&>(f1),
static_cast<FigRectangle&>(f2));
    }
}
void MmCircCircSpecOut(ofstream& ofst, Figure& f1, Figure& f2) {
    if(f1.mark == GetRegMarkFigCircle() && f2.mark == GetRegMarkFigCircle()) {
        return CircCircOut(ofst, static_cast<FigCircle&>(f1),
static_cast<FigCircle&>(f2));
    }
}
namespace {
    class Register {
    public:
        Register(const char* regInfo);
    };
    Register::Register(const char* regInfo) {
        cout << regInfo << endl;
multimethodFunc[GetRegMarkFigTriangle()][GetRegMarkFigCircle()] =
MmTrianCircSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigTriangle() << "][" <<
GetRegMarkFigCircle() << "] = MmTrianCircSpecOut" << endl;
multimethodFunc[GetRegMarkFigRectangle()][GetRegMarkFigCircle()] =
MmRectCircSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigRectangle() << "][" <<
GetRegMarkFigCircle() << "] = MmRectCircSpecOut" << endl;
multimethodFunc[GetRegMarkFigCircle()][GetRegMarkFigTriangle()] =
MmCircTrianSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigCircle() << "][" <<
GetRegMarkFigTriangle() << "] = MmCircTrianSpecOut" << endl;
multimethodFunc[GetRegMarkFigCircle()][GetRegMarkFigRectangle()] =
MmCircRectSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigCircle() << "][" <<
GetRegMarkFigRectangle() << "] = MmCircRectSpecOut" << endl;
multimethodFunc[GetRegMarkFigCircle()][GetRegMarkFigCircle()] =
MmCircCircSpecOut;
        cout << "    multimethodFunc[" << GetRegMarkFigCircle() << "][" <<
GetRegMarkFigCircle() << "] = MmCircCircSpecOut" << endl;
    }
    Register trianRegisterPerimeter("Registration of: Extended combinations of
Triangle, Rectangle, Circle in  Multimethod");
}
```

ПРИЛОЖЕНИЕ К

Библиотека макроопределений

dclGeneric.h – Реализация макросов, формирующих декларации для обобщений в заголовочных файлах

```
// Формирование структуры обобщения. Заголовок описания.
#define DECLARE_GENERIC(GenName, ...)\
    struct GenName {\
    int _mark_; \
    __VA_ARGS__ \
    }; \
    int GetNextMarkAndIncrement##GenName(); \
    int GetMark##GenName();
// Макросы, формирующие описания указателей на обобщенные функции и заголовки
этих обобщенных функций
#define DECLARE_GENERIC_FUN(GenFunName, resultType, ...) \
    typedef resultType (*GenFunName ## Func)(__VA_ARGS__); \
    extern GenFunName  ## Func _g ## GenFunName[]; \
    resultType GenFunName(__VA_ARGS__);
```

CreateSpecialization.h – Реализация макроса, формирующего основу специализации

```
#define CREATE_SPECIALIZATION(Name, BaseName, SpecName) \
    struct Name : BaseName { \
        SpecName _spec; \
    };
```

RegisterSpecialization.h – Реализация макроса регистрации специализации в специальном параметрическом массиве

```
#include "ClassMarkRegistrar.h"
#define REGISTER_SPECIALIZATION(SpecName, IncrFunc, DebugInfo) \
    namespace \
    { \
        void InitRegMark##SpecName() \
        { \
            regMark##SpecName = IncrFunc(); \
        } \
        ClassMarkRegistrar reg##SpecName(InitRegMark##SpecName, DebugInfo); \
    }
```

MethodRegistrar.h – Вспомогательный файл для макроса регистрации обработчиков специализаций

```
#include <iostream>
class MethodRegistrar
{
public:
    template<typename TMethod>
    MethodRegistrar(
            TMethod container[],
            TMethod method,
            int index,
            const char* info = nullptr)
    {
        if (info != nullptr) std::cout << info << '\n';
        container[index] = method;
    }
};
```

RegisterMethod.h – Макрос, регистрирующий обработчик специализации в параметрическом массиве

```
#include "MethodRegistrar.h"
#define REGISTER_METHOD(Container, Method, Mark, DebugInfo) \
    MethodRegistrar regMethod##Method(Container, Method, Mark, DebugInfo);
```

ClassMarkRegistrar.h – Вспомогательный файл для макроса регистрации специализаций

```
#include <iostream>
int GetSpecNumAndIncrement();
class ClassMarkRegistrar
{
public:
    typedef void(*Initializer)();
    explicit ClassMarkRegistrar(Initializer init, const char* regInfo = nullptr)
    {
        init();
        if (regInfo != nullptr) std::cout << regInfo << std::endl;
    }
};
```

CreateRegMarkMethod.h – Макрос создания специализации (выделения ей маркера)

```
#define CREATE_REG_MARK_METHOD(SpecName) \
    namespace { \
        int regMark##SpecName = -1; \
        int GetRegMark##SpecName() \
        { \
            return regMark##SpecName; \
        } \
    }
```